



Deadlock and starvation free reentrant readers–writers: A case study combining model checking with theorem proving[☆]

Bernard van Gastel^a, Leonard Lensink^a, Sjaak Smetsers^a, Marko van Eekelen^{a,b,*}

^a Institute for Computing and Information Sciences, Radboud University Nijmegen, Netherlands

^b School of Computer Science, Open University of the Netherlands, Netherlands

ARTICLE INFO

Article history:

Received 31 May 2009

Accepted 9 March 2010

Available online 16 June 2010

Keywords:

Model checking

Theorem proving

Readers–writers algorithm

SPIN

PVS

ABSTRACT

The classic readers–writers problem has been extensively studied. This holds to a lesser degree for the reentrant version, where it is allowed to nest locking actions. Such nesting is useful when a library is created with various procedures each starting and ending with a lock operation. Allowing nesting makes it possible for these procedures to call each other.

We considered an existing widely used industrial implementation of the reentrant readers–writers problem. Staying close to the original code, we modelled and analyzed it using a model checker resulting in the detection of a serious error: a possible deadlock situation. The model was improved and checked satisfactorily for a fixed number of processes. To achieve a correctness result for an arbitrary number of processes the model was converted to a specification that was proven with a theorem prover. Furthermore, we studied starvation. Using model checking we found a starvation problem. We have fixed the problem and checked the solution. Combining model checking with theorem proving appeared to be very effective in reducing the time of the verification process.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

It is generally acknowledged that the historical growth in processor speed is reaching a hard physical limitation. This has led to a revival of interest in concurrent processing. Also in industrial software, concurrency is increasingly used to improve efficiency [30]. It is notoriously hard to write correct concurrent software. Finding bugs in concurrent software and proving the correctness of (parts of) this software is therefore attracting more and more attention, in particular where the software is in the core of safety critical or industrial critical applications.

However, it can be incredibly difficult to track down concurrent software bugs. In concurrent software, bugs are typically caused by infrequent ‘race conditions’ that are hard to reproduce. In such cases, it is necessary to thoroughly investigate ‘suspicious’ parts of the system in order to improve these components in such a way that correctness is guaranteed.

Three commonly used techniques for checking correctness of such a system are *testing*, *static (code) analysis* and *formal verification*. In practice, testing is widely and successfully used to discover faulty behaviour, but it cannot assure the absence of bugs. In particular, for concurrent software testing is less suited due to the typical characteristics of the bugs (infrequent and hard to reproduce). In contrast with testing, static analysis is performed directly and fully automatically on the source code, without actually executing it. The information obtained from the analysis are, for example, common coding errors and

[☆] This paper is an extended version of van Gastel et al. (2009) [13] which received the EASST Best Software Science Paper Award of FMICS2008.

* Corresponding author at: Institute for Computing and Information Sciences, Radboud University Nijmegen, Netherlands.

E-mail addresses: b.vangastel@cs.ru.nl (B. van Gastel), l.lensink@cs.ru.nl (L. Lensink), s.smetsers@cs.ru.nl (S. Smetsers), marko@cs.ru.nl, m.vaneekelen@cs.ru.nl (M. van Eekelen).

suspicious control flow (e.g. leading to null pointer exceptions or lock order violations). There are roughly two approaches to formal verification: *model checking* and *theorem proving*. Model checking [7,26] has the advantage that it can be performed automatically, provided that a suitable model of the software (or hardware) component has been created. Furthermore, in the case a bug is found model checking yields a counterexample scenario. A drawback of model checking is that it suffers from the state-space explosion problem and typically requires a closed system. In principle, theorem proving can handle any system. However, creating a proof may be hard and it generally requires a large investment of time. It is only partially automated and mainly driven by the user's understanding of the system. Besides, when theorem proving fails this does not necessarily imply that a bug is present. It may also be that the proof could not be found by the user.

We will consider the *reentrant readers–writers* problem as a formal verification case study. The classic readers–writers problem [9] considers multiple processes that want to have read and/or write access to a common resource (a global variable or a shared object). The problem is to set up an access protocol such that no two writers are writing at the same time and no reader is accessing the common resource while a writer is accessing it. The classic problem is studied extensively [25]; the reentrant variant (in which locking can be nested) has received less attention so far although it is used in Java, C# and C++ libraries.

We have chosen a widely used industrial C++ library (Trolltech's Qt) that provides methods for reentrant readers–writers. For this library a serious bug is revealed and removed. This case study is performed in a structured manner combining the use of a model checker with the use of a theorem prover exploiting the advantages of these methods and avoiding their weaknesses. The main achievement of this approach is that it significantly improves the time effectiveness of the verification process itself.

This paper can be seen as an extended version of [13]. There are two main differences. Firstly, in this version we managed to keep the model much *closer to the code* using PROMELA and SPIN in stead of Uppaal. The model contains more of the details present in the C++ program and it looks like the C++ program, but is still at approximately the same abstraction level as the model in [13]. We have *manually* translated both the original C++ code into SPIN models and the SPIN models into PVS specifications. However, by keeping the model and the specification so close to the C++ code, we have shown that our approach lends itself for tool support, i.e. the used translations indicate ways of performing the conversion in a (semi) automatic way. Secondly, in this paper we also studied *starvation*.

In Section 2 we will introduce the abstract readers–writers problem. The studied Qt implementation is discussed in Section 3. Its model will be defined, improved and checked for a fixed number of processes in Section 4. Using a theorem prover the model will be fully verified in Section 5. Finally, related work, future work and concluding remarks are found in Sections 6 and 7.

2. The readers–writers problem

If in a concurrent setting two threads are working on the same resource, synchronisation of operations is often necessary to avoid errors. A *test-and-set* operation is an important primitive for protecting common resources. This atomic (i.e. non-interruptible) instruction is used to both test and (conditionally) write to a memory location. To ensure that only one thread is able to access a resource at a given time, these processes usually share a global boolean variable that is controlled via *test-and-set* operations, and if a process is currently performing a test-and-set, it is guaranteed that no other process may begin another test-and-set until the first process is done. This primitive operation can be used to implement *locks*. A lock has two operations: lock and unlock. The lock operation is done before the critical section is entered, and the unlock operation is performed after the critical section is left. However, implementing a lock with just an atomic test-and-set operation is impracticable. More realistic solutions will require support of the underlying OS: threads acquiring a lock already occupied by some thread should be de-scheduled until the lock is released. A variant of this way of locking is called *condition locking*: a thread can wait until a certain condition is satisfied, and will automatically continue when notified (*signalled*) that the condition has been changed. An extension for both basic and condition locking is *reentrancy*, i.e. allowing nested lock operations by the same thread.

A so-called *read–write* lock functions differently from a normal lock: it either allows multiple threads to access the resource in a read-only way, or it allows one, and only one, thread at any given time to have full access (both read and write) to the resource [14]. These locks are used in databases and file systems.

Several kinds of solutions to the classical readers–writers problem exist. Here, we will consider a *read–write* locking mechanism with the following properties.

writers preference. Most solutions give priority to write locks over read locks because write locks are assumed to be more important, smaller, exclusive, and to occur less frequently. The main disadvantage of this choice is that it results in the possibility of reader starvation: when constantly there is a thread waiting to acquire a write lock, threads waiting for a read lock will never be able to proceed.

reentrant. A thread can acquire the lock multiple times, even when the thread has not fully released the lock. Note that this property is important for modular programming: a function holding a lock can use other functions which possibly acquire the same lock. We distinguish two variants of reentrancy:

```

struct QReadWriteLockPrivate {
    QReadWriteLockPrivate()
    : accessCount(0),
      currentWriter(0),
      waitingReaders(0),
      waitingWriters(0)
    { }

    QMutex mutex;
    QWaitCondition readerWait,
                    writerWait;

    Qt::HANDLE currentWriter;
    int accessCount, waitingReaders,
    waitingWriters;
};

void QReadWriteLock::lockForRead() {
    QMutexLocker lock(&d->mutex);
    while (d->accessCount < 0 ||
           d->waitingWriters) {
        ++d->waitingReaders;
        d->readerWait.wait(&d->mutex);
        --d->waitingReaders;
    }
    ++d->accessCount;
    Q_ASSERT_X(d->accessCount>0,
               "...", "...");
}

void QReadWriteLock::lockForWrite() {
    QMutexLocker lock(&d->mutex);
    Qt::HANDLE self =
        QThread::currentThreadId();
    while (d->accessCount != 0) {
        if (d->accessCount < 0 &&
            self == d->currentWriter) {
            break; // recursive write lock
        }
        ++d->waitingWriters;
        d->writerWait.wait(&d->mutex);
        --d->waitingWriters;
    }
    d->currentWriter = self;
    --d->accessCount;
    Q_ASSERT_X(d->accessCount<0,
               "...", "...");
}

void QReadWriteLock::unlock() {
    QMutexLocker lock(&d->mutex);
    Q_ASSERT_X(d->accessCount!=0,
               "...", "...");
    if ((d->accessCount > 0 &&
         --d->accessCount == 0) ||
        (d->accessCount < 0 &&
         ++d->accessCount == 0)) {
        d->currentWriter = 0;
        if (d->waitingWriters) {
            d->writerWait.wakeOne();
        } else if (d->waitingReaders) {
            d->readerWait.wakeAll();
        }
    }
}

```

Fig. 1. The QReadWriteLock class of Qt 4.3.

1. *Weakly reentrant*: only permit sequences of either read or write locks;
2. *Strongly reentrant*: permit a thread holding a write lock to acquire a read lock. This will allow the following sequence of lock operations: write_lock, read_lock, unlock, unlock. Note that the same function is called to unlock both a write lock and a read lock. The sequence of a read lock followed by a write lock is not permitted because of the evident risk of a deadlock (e.g. when two threads both want to perform the locking sequence read_lock, write_lock they can both read but none of them can write).

3. Qt's implementation of readers–writers locks

In this section we show the C++ implementation of weakly reentrant read–write locks being part of the multi-threading library of the Qt development framework, version 4.3. The code is not complete; parts that are not relevant to this presentation are omitted. This implementation uses other parts of the library: threads, mutexes and conditions. Like e.g. in Java, a condition object allows a thread that owns the lock but that cannot proceed, to wait until some condition is satisfied. When a running thread completes a task and determines that a waiting thread can now continue, it can call a signal on the corresponding condition. This mechanism is used in the C++ code listed in Fig. 1.

The structure QReadWriteLockPrivate contains the attributes of the class QReadWriteLock. These attributes are accessible via an indirection named d. The attributes mutex (of type QMutex), readerWait (of type QWaitCondition) and writerWait (of type QWaitCondition) are used to synchronize access to the other administrative attributes, of which accessCount keeps track of the number of locks acquired (including reentrant locks) for this lock. A negative value is used for write access and a positive value for read access. The attributes waitingReaders and waitingWriters (both int's) indicate the number of threads requesting a read respectively write permission, that are currently pending. If some thread owns the write lock, currentWriter contains a HANDLE to this thread; otherwise currentWriter is a null pointer.

The code itself is fairly straightforward. The locking of the mutex is done via the constructor of the wrapper class QMutexLocker. Unlocking this mutex happens implicitly in the destructor of this wrapper. Observe that a write lock can only be obtained when the lock is completely released (d->accessCount == 0), or the thread already has obtained a write lock (a reentrant write lock request, d->currentWriter == self).

The code could be polished a bit. E.g. one of the administrative attributes can be expressed in terms of the others. However, we have chosen not to deviate from the original code, except for the messages in the assertions which were, of course, more informative.

```

typedef pthread_mutex_t {
    bool locked = false
};

5 inline pthread_mutex_unlock(this) {
    assert(this.locked);
    this.locked = false;
}

                                inline pthread_mutex_lock(this) {
10     atomic {
        !this.locked;
        this.locked = true;
    }
15 }

```

Fig. 2. Abstract model in PROMELA of the non-reentrant pthread_mutex.

4. Model checking readers–writers with SPIN

SPIN is an explicit state model checker with support for assertions and Linear Temporal Logic (LTL), including liveness properties. SPIN converts a model written in the specification language PROMELA to a checker written in C. By compiling and running the checker, properties can be checked; e.g. see [18,5].

In the previous version of this paper [13] we used Uppaal for modelling the system. An advantage of Uppaal is its intuitive and easy to use graphical interface. However, we have decided to switch to SPIN for mainly two reasons: First, the input language PROMELA resembles C, which allows us to model the code in a direct and clear way. Second, compiled models generated by SPIN appear to be more efficient than equivalent models specified in Uppaal. This enables us to enlarge the examined state space of the model significantly.

A few general notes can be made about modelling code in PROMELA. PROMELA is not a (general-purpose) programming language, and therefore it lacks some features that are found in common language like C or JAVA. For instance, there are no functions that return values in PROMELA. For simple non-recursive procedures, one can use the `inline` construct instead. Moreover, PROMELA does not support object oriented programming. In our translation, we will represent the attributes of objects as structs, and non-static methods as (inline) functions, having `this` as an explicit argument.

A feature of SPIN is the ability to embed C code directly. With a couple of special PROMELA statements C code can be inserted in the model and is executed atomically in the model. SPIN tracks the memory used by these statements and include the memory regions in the state space. One can easily convert source code to a PROMELA model by wrapping all C code in the proper PROMELA statements. This method is not applicable to our case study: the mutexes are system calls which modify memory outside the process space. The content of these (kernel) memory regions cannot be rolled back by SPIN as the state space is explored. So we have to model the whole program in PROMELA.

4.1. Modelling the basics

The Qt implementation of the QReadWriteLock class is based on two other classes: QMutex and QWaitCondition. These components are platform dependent. In our case study we use the Linux version, in which QMutex and QWaitCondition are built on the pthread_mutex and pthread_cond components of the POSIX Thread Library. This library is part of the operating system. Creating a code based model of these components would require the treatment of OS dependent details making the whole system too complex. Instead we will use abstract versions of these components.

When using the 2.6 version of the Linux kernel, the default behaviour for POSIX components is not starvation free. Starvation free behaviour of these components can be activated by setting the SCHED_FIFO flag when creating threads. Qt, however, uses the default behaviour. This is, of course, an important observation when we are considering the absence of starvation of the locking mechanism. In that case we will assume that the threads are scheduled fairly and that the underlying basic locking primitives use a first-in first-out (FIFO) lock assignment strategy, see Section 4.6. However, below we study the default behaviour of the POSIX components first.

We start with modelling the basic pthread_mutex class. The two main functions of this component are pthread_mutex_lock and pthread_mutex_unlock, which both can be specified easily in PROMELA; see Fig. 2. The lock itself is represented as a single boolean (named locked), initially set to false. The pthread_mutex_lock function is an atomic operation that waits until locked is false before setting it to true. Waiting can be expressed in PROMELA just by using boolean expressions as statements. If, during the execution of the model such a statement is encountered, the corresponding computation branch will be suspended until the expression has become true. The pthread_mutex_unlock function resets locked to false. To check for incorrect use, an assertion is added to the code verifying that no lock is released if it has not been obtained before. By wrapping the locked variable in a typedef (named pthread_mutex_t) we can use this pthread_mutex component in the same manner as in the original C++ code.

We now model pthread_cond. This component allows a thread owning the lock to wait until some condition is satisfied (while releasing the lock). When another running thread completes a task and determines that a waiting thread can now continue, it can wake up this thread by calling a signal on the corresponding condition. Actually, two kinds of signals are available in pthread_cond: pthread_cond_signal (waking one thread) and pthread_cond_broadcast (waking all threads). Our abstract version of pthread_cond uses a basic synchronisation mechanism of PROMELA: (synchronous) rendezvous channels. The pthread_cond_wait function uses a send operation on the rendezvous channel cont.

```

typedef pthread_cond_t {
    byte waiting = 0;
    chan cont = [0] of {bit};
};
5
inline pthread_cond_signal(this) {
    atomic {
        if
            :: this.waiting > 0 ->
10         this.waiting--;
            this.cont?_;
        :: else
            fi;
    }
15 }

inline pthread_cond_broadcast(this) {
    atomic {
20     do
        :: this.waiting > 0 ->
            this.waiting--;
            this.cont?_;
        :: else -> break;
25     od;
    }
}

inline pthread_cond_wait(this,mutex) {
30     this.waiting++;
    pthread_mutex_unlock(mutex);
    this.cont!1;
    pthread_mutex_lock(mutex);
}

```

Fig. 3. Abstract model in PROMELA of pthread_cond.

```

typedef QWaitCondition {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
5    int waiters = 0;
    int wakeups = 0;
};

inline QWaitCondition_wakeOne(this) {
    pthread_mutex_lock(this.mutex);
10    this.wakeups = min(this.wakeups + 1,
        this.waiters);
    pthread_cond_signal(this.cond);
    pthread_mutex_unlock(this.mutex);
}

15
inline QWaitCondition_wakeAll(this) {
    pthread_mutex_lock(this.mutex);
    this.wakeups = this.waiters;
    pthread_cond_broadcast(this.cond);
20    pthread_mutex_unlock(this.mutex);
}

inline QWaitCondition_wait(this, m) {
    pthread_mutex_lock(this.mutex);
25    this.waiters++;
    QMutex_unlock(m);
    do
        :: this.wakeups == 0 ->
            pthread_cond_wait(this.cond,
                this.mutex);
30    :: else ->
        break;
    od;
    this.waiters--;
    this.wakeups--;
35    pthread_mutex_unlock(this.mutex);
    QMutex_lock(m);
}

```

Fig. 4. Concrete model in PROMELA of QWaitCondition.

The thread invoking this method will be blocked until another thread execute a receive operation. The contents of the message sent over this channel are irrelevant, only the timing of the message counts. On the receiver side this is specified by using an anonymous write-only variable (in PROMELA: `cont?_`), and on the sender side by choosing some arbitrary value (in our case the value 1, sent with the statement `cont!1`). Before waiting on the channel the wait function has to unlock the mutex and, after continuing, to lock the mutex again. To be able to wake all the waiting threads, the condition keeps track of the number of waiting threads in the variable `waiting`. For correctness atomic blocks are used to limit the interleaving of processes (otherwise the test `waiting > 0` and `waiting--` could be interrupted). Just like `pthread_mutex` the variables are wrapped in a new type `pthread_cond_t`. The model is listed in Fig. 3.

The implementation of `QMutex` class appears to be rather complex, due to some optimisations that have been performed. As a consequence, the code base is large and it is outside the scope of this article, to model this part faithfully. Instead we will use `pthread_mutex` to provide the locking mechanism, because it has the same functional behaviour as `QMutex`. Hence `QMutex` is a wrapper around `pthread_mutex`. The implementation of `QWaitCondition`, on the other hand, is much shorter, and can therefore be converted to PROMELA straightforwardly. The result is listed in Fig. 4. Again, the attributes of this class are wrapped in a struct. As one can see, the class depends on `pthread_mutex`, `pthread_cond` (appearing as attribute types), and on `QMutex` (passed as an argument to the method `QWaitCondition_wait`). According to the comments in the source code ‘many vendors warn of spurious wake-ups from `pthread_cond_wait`, especially after signal delivery’. Both the variable `wakeups` and the loop in `wait` method are used to counter the described spurious wake-ups. In this way, a thread can only finish the `wait` method if a signal is received. The variable `wakeups` is used to keep track of the number of threads allowed to wake up and is bound by the number of waiting threads, as contained in the variable `waiters`. Both the `wakeOne` and the `wakeAll` methods increase the `wakeups` variable, and the `wait` method decreases the variable as threads are woken. The `pthread_mutex` used in `QWaitCondition` is needed because `QMutex` does not use a `pthread_mutex`, and such a mutex is needed for the `pthread_cond_wait` function. The parameter `m` of the `wait` method is a mutex. This mutex is released until a signal is received.

```

struct QReadWriteLockPrivate {
    QMutex mutex;
    QWaitCondition readerWait,
        writerWait;
5   Qt::HANDLE currentWriter;
    int accessCount,
        waitingReaders,
        waitingWriters;
};
10 void QReadWriteLock::lockForRead() {
    QMutexLocker lock(&d->mutex);
    while (d->accessCount < 0 ||
15         d->waitingWriters) {
        ++d->waitingReaders;
        d->readerWait.wait(&d->mutex);
        --d->waitingReaders;
    }
    ++d->accessCount;
20   Q_ASSERT_X(d->accessCount > 0,
               "...", "...");
}

typedef QReadWriteLock {
    QMutex mutex;
    QWaitCondition readerWait;
    QWaitCondition writerWait;
5   pid currentWriter = NT;
    int accessCount = 0;
    int waitingReaders = 0;
    int waitingWriters = 0;
};
10 inline QReadWriteLock_lockForRead(this) {
    QMutex_lock(this.mutex);
    do
        :: this.accessCount < 0 ||
15         this.waitingWriters > 0 ->
            this.waitingReaders++;
            QWaitCondition_wait(this.readerWait,
                               this.mutex);
            this.waitingReaders--;
20   :: else -> break;
    od;
    this.accessCount = this.accessCount + 1;
    assert(this.accessCount > 0);
    QMutex_unlock(this.mutex);
25 }

```

Fig. 5. Part of QReadWriteLock (QT 4.3 version) in C++ (left) and PROMELA (right).

4.2. Modelling readers–writers

Now we have modelled all the components on which the QReadWriteLock class depends, we can convert the QReadWriteLock itself to PROMELA. All class attributes can be expressed directly (the type Qt::HANDLE is converted to the PROMELA type pid, both identifying a specific process or thread). In Fig. 5 the variables of the class and the code of lockForRead are listed, on the left the original C++ code, and on the right the conversion in PROMELA. Methods are converted to inline definitions.

The QMutexLocker is a convenience wrapper around a lock, obtaining a lock when the object is constructed and releasing the lock implicitly (via its destructor) when the object is deallocated. When used as a local (stack) object, QMutexLocker obtains the lock during its initialisation and releases the lock when this local object gets out of scope. This implicit destructor invocation is converted to an explicit call of QMutexUnlock.

The translation of the code for the lockForRead method is performed instruction-wise. A while-loop is converted into a do . . . od statement (which can be thought of as for (; ;) in C++). The loop is ended with a break in one of the *condition blocks* (statements prefixed by ::). Normally, a block with a true condition is chosen non-deterministically for execution, though in our case only one of these conditions can possibly hold at a given time. The rest of the PROMELA code should be self-explanatory.

4.3. Modelling usage of the lock

In order to check properties we will simulate all possible usages of the QReadWriteLock. For this reason we will define a number of threads, each (sequentially) executing a finite number of read and/or write locks, and matching unlocks, in a proper sequence (i.e. no unlocks if the lock is not obtained first by the thread and no write lock requests if the thread already has a read lock). Eventually each thread relinquishes all locks, so other threads are allowed to proceed. The variable maxLocks indicates how many locks a thread may request before it relinquishes all locks. We model these threads by PROMELA processes as shown in Fig. 6. Here, THREADS indicates the number of threads the model is checked with. Note that the do statement chooses one of the options non-deterministically. The readNest variable is used to exclude the case in which a (reentrant) write lock is performed after a read lock is already obtained. Both readNest and writeNest are used to control unlocking. Both are updated in the ‘methods’ of QReadWriteLock. As the ‘methods’ are in fact just inlined code, they can access and update these variables.

There are three kinds of properties to be checked, each invoked differently by SPIN. The absence of deadlock property is checked implicitly when running the verifier for assertion violations. Each time a non-end state is encountered and no transitions out of the state are valid an ‘invalid end state’ error is reported. The second type of properties we check are safety properties, which are valid in each state of the model (specified as LTL formulas beginning with the [] operator). Most of the informal correctness properties specified in Section 2 are of this type. The last type are liveness properties, guaranteeing that each process can make progress of some sort. SPIN has special support for liveness properties, called *progress states*, but they can also be checked with LTL properties. We continue with checking for deadlock and assertions.


```

active[THREADS] proctype user() {
    int readNest = 0;
    int writeNest = 0;
    int maxLocks;
5   do
        :: maxLocks = MAXLOCKS;
        do
            :: maxLocks > 0 ->
                maxLocks--;
10        if
            :: readNest == 0 -> QReadWriteLock_lockForWrite(rwlock);
            :: QReadWriteLock_lockForRead(rwlock);
            fi;
            :: writeNest + readNest > 0 ->
15        QReadWriteLock_unlock(rwlock);
            :: maxLocks != MAXLOCKS && writeNest + readNest == 0 ->
                break;
        od;
    od;
20 }

```

Fig. 6. PROMELA process of QReadWriteLock usage.

```

pan: invalid end state (at depth 188)
pan: wrote qreadwritelock43.usage.trail
...
pan: reducing search depth to 32
5 ...
    0: enter lockForRead
    0: leave lockForRead
    1: enter lockForWrite
    1: waiting
10  0: enter lockForRead
    0: waiting
spin: trail ends after 34 steps
#processes: 2
    rwlock.mutex.m.lockedBy = 255
    rwlock.mutex.m.count = 0
15  rwlock.readerWait.waiters = 1
    rwlock.readerWait.wakeups = 0
    rwlock.readerWait.waiting = 1
    rwlock.writerWait.waiters = 1
20  rwlock.writerWait.wakeups = 0
    rwlock.writerWait.waiting = 1
    rwlock.accessCount = 1
    rwlock.currentWriter = 255
    rwlock.waitingReaders = 1
25  rwlock.waitingWriters = 1
    readers = 1
    writers = 0
34:  proc  0 (user) line  19 "qwaitcondition.abs" (state 29)
34:  proc  1 (user) line  19 "qwaitcondition.abs" (state 187)

```

Fig. 7. Output of SPIN when checking for a deadlock.

4.4. Checking for deadlock and assertions

As stated before, deadlock detection is done implicitly when checking for assertions. Each state not marked as an end state and with no outgoing transitions is reported. Also all assertions in the model are checked. Besides the assertions that were present in the original code, there is one assertion in `lockForWrite()` that has been added, to verify that no thread gets write access when readers are busy.

Running our model resulted immediately in the detection of a deadlock. The output of SPIN is given in Fig. 7. It starts with an iterative search for the shortest error trail. After that the debug output of the shortest trail is printed. The values of all variables in the last state are showed, and the output ends with a message in which state the processes are. The situation reported by SPIN occurs when a thread already having a read lock requests another one, while another thread is waiting for a write lock. The deadlock is clear: the first thread is never going to proceed with the reentrant reader because there is a writer waiting. The second thread is never going to proceed because the lock is never released. A change to the algorithm is needed to avoid this deadlock.

The solution to the deadlock stated above is to let a reentrant lock always proceed. To check if a lock request is a reentrant operation, for each thread the number of calls to the specific lock should be kept track of. If this number is positive the lock operation should always succeed. In the original C++ code, an extra variable `count` of type `QHash<Qt::HANDLE, int>` is introduced, mapping thread identifiers to numbers. In our translated model we represented this hash table by an integer

```

typedef QReadWriteLock {
    QMutex mutex;
    QWaitCondition readerWait;
    QWaitCondition writerWait;
5
    int threadCount = 0;
    int waitingReaders = 0;
    int waitingWriters = 0;

10
    pid currentWriter = NT;
    int count[THREADS] = 0;
}

inline QReadWriteLock_lockForRead(this) {
15
    QMutex_lock(this.mutex);
    // check if this is a reentrant lock
    if
    :: this.count[_pid] == 0 ->
    do
20
        :: (this.currentWriter != NT ||
           this.waitingWriters > 0) ->
           this.waitingReaders++;
           QWaitCondition_wait
           (this.readerWait, this.mutex);
25
           this.waitingReaders--;
           :: else -> break;
           od;
           this.threadCount++;
           assert(this.waitingWriters == 0);
30
           :: else
           fi;
           this.count[_pid]++;
           ... update model variables ...
           QMutex_unlock(this.mutex);
35
}

inline QReadWriteLock_lockForWrite(this) {
    QMutex_lock(this.mutex);
    // check if this is a reentrant lock
40
    if
    :: this.currentWriter != _pid ->
    do
        :: this.threadCount != 0 ->
           this.waitingWriters++;
           QWaitCondition_wait
45
           (this.writerWait, this.mutex);
           this.waitingWriters--;
           :: else -> break;
           od;
           this.currentWriter = _pid;
           this.threadCount++;
           assert(this.count[_pid] == 1 &&
50
           this.currentWriter == _pid);
           this.count[_pid]++;
           ... update model variables ...
           QMutex_unlock(this.mutex);
           }

60
inline QReadWriteLock_unlock(this) {
    QMutex_lock(this.mutex);
    this.count[_pid]--;
    // is it the last unlock by this thread?
65
    if
    :: this.count[_pid] == 0 ->
       this.threadCount--;
       // is it the last unlock of the lock?
       if
70
       :: this.threadCount == 0 ->
          this.currentWriter = NT;
          if
          // if available wake one writer,
          :: this.waitingWriters > 0 ->
             QWaitCondition_wakeOne
             (this.writerWait);
          // otherwise wake all readers
          :: else ->
             if
80
             :: this.waitingReaders > 0 ->
                QWaitCondition_wakeAll
                (this.readerWait);
             :: else
             fi;
85
             fi;
             :: else
             fi;
             :: else
             fi;
90
             ... update model variables ...
             QMutex_unlock(this.mutex);
             }
}

```

Fig. 8. Updated PROMELA model of readers–writers algorithm.

array count in which count [pid] is the number of reentrant locks of process pid. In PROMELA the array is declared with the statement `int count [THREADS]`.

Furthermore, we take this opportunity to change the strange use of the accessCount variable: the sign of the value of accessCount indicates whether active locks are read locks or write locks. This distinction between readers and writers appears to be superfluous. In fact, leaving out this distinction provides that our implementation is strongly reentrant. Moreover, we changed the name of the variable into threadCount to indicate it actually contains the number of different threads that are currently holding the lock.

After the adjustments to the model, SPIN reports no assertion violations and no invalid end states for a parameterised model with three threads and a maximum of five locking operations. So the model is shown to be free of deadlocks with these parameters.

We reported the deadlock to Trolltech. Recently, Trolltech released a new version of the thread library (version 4.4) in which the deadlock was repaired. However, the new version of the Qt library is still only weakly reentrant, not admitting threads that have write access to do a read lock. This limitation unnecessarily hampers modular programming.

4.5. Checking LTL safety properties

To check the properties we introduce auxiliary variables in the model to track the number of threads having write locks (called writers) and having read locks (called readers). The code needed to keep track of these auxiliary variables is inserted at appropriate place in the ‘methods’ of QReadWriteLock. The readers and writers variables are only incremented on a non-reentrant call of a thread, and therefore decremented only on the final unlock. The other variables stated in the properties are attributes of QReadWriteLock.

We now continue with checking LTL safety properties of the algorithm. These properties are checked by querying SPIN with a LTL expression. We removed a deadlock in the previous subsection, but the algorithm was not checked for conceptually flawed behaviour, for example allowing both a reader and a writer enter the critical section at the same time. A predicate called `outsideCS` is introduced, indicating that *no* change can occur inside the lock structure. In other words no thread has locked the mutex, as indicated by the negation of the boolean attribute `mutex.locked` from `QReadWriteLock`.

Formalisation of the properties stated in Section 2 is now straightforward. The resulting invariants are listed below. The `waitingReaders` and `waitingWriters` variables used are attributes from the `QReadWriteLock` object.

- $[] (\text{readers} = 0 \vee \text{writers} = 0)$
There are not simultaneously writers *and* readers allowed.
- $[] (\text{writers} \leq 1)$
No more than one writer is allowed.
- $[] (\text{outsideCS} \rightarrow (\text{waitingWriters} > 0 \rightarrow (\text{readers} > 0 \vee \text{writers} > 0)))$
States that the only possibility of waiting writers is when there are readers or writers busy, but only when there is no change to the lock.
- $[] (\text{outsideCS} \rightarrow (\text{waitingReaders} > 0 \rightarrow (\text{writers} > 0 \vee \text{waitingWriters} > 0)))$
States that the only possibility of waiting readers is when there are writers waiting or writers busy, but only when there is no change to the lock.

The third and fourth invariant do not hold for this algorithm. We detected this issue during model checking. There exists a state in which the proposition `outsideCS` is true, there are no readers and no writers, but there are readers and/or writers waiting. The third and fourth stated safety property are therefore violated. This occurs if a thread has just called the `unlock` method, and another thread intends to continue with acquiring a read or a write lock. The invariants are not easily fixed, as these states cannot be easily excluded. In the next subsection, a change is proposed to avoid starvation. This change also avoids the state mentioned above. Therefore we postpone verifying these invariants to the next subsection.

4.6. Checking for absence of starvation

We continue with ensuring the absence of starvation in the algorithm. In Section 2 we stated that the design decision to give preference to writers results in a possible reader starvation. Therefore it only makes sense to check the property for writers. In SPIN one can verify starvation properties by using *progress states*. A looping process obtaining and releasing write locks, but no read locks, is added and labelled with a *progress label*. When checking the model, it is verified that all execution cycles (i.e. an execution path on which the same state occurs twice) contain this progress label.

As noted earlier, the original readers–writers algorithm has a starvation problem because Qt uses the default behaviour of POSIX on Linux. However, we continue as if a fair scheduling policy would have been used. To avoid starvation in the underlying `pthread_mutex` and `pthread_cond` models, these were replaced by starvation free versions that use a FIFO mechanism. Despite of these changes, the model still contains the possibility of writers starvation. This appeared when we checked the model for absence of progress, and SPIN found an execution cycle with no progress states. A graphic representation of this cycle is shown in Fig. 9.

The problem is caused by the `wait` method of `QWaitCondition`; see Fig. 4. When thread t calls `QWaitCondition_wait`, it will suspend execution (by calling `pthread_cond_wait`) until thread s signals that thread t can continue its execution. However, at that time t has no longer locked the mutex `this.mutex`. Each other thread, thread s in the figure, can now lock this mutex (by calling `lockForWrite`) just before t does, effectively stealing the turn of t .

This problem can be avoided by ensuring that no thread can get the mutex before the signalled thread (t in the above example) can start executing again. This can be done by atomically transferring the lock on the mutex from the signalling thread to the signalled thread. Also, all stated invariants are valid, as the states mentioned in the previous subsection do not exist anymore because of the atomic transfer of the lock between threads. To accommodate this behaviour we have adjusted the `QWaitCondition` and `QMutex` parts of our SPIN model. Although we were able to find a solution, the solution is rather large and complex. The solution also includes a way to create a starvation free condition variables out of one starvation free mutex and two starvation-prone condition variables. This is needed because starvation free condition variables are not available on most POSIX platforms, including Linux, Mac OS X and FreeBSD. Due to space limitations, we will not present the adjusted SPIN model, but take the improvement into account in the next section. For the complete solution and a more extensive report of our experiments, see [12]. The adjusted version is verified free of deadlock and starvation and not violating the safety properties, for a model with three threads with a maximum of four lock operations (actually we were able to verify the model free of starvation for three threads and a maximum of six reentrant lock operations, but the other properties only for a model with a maximum of four reentrant lock operations).

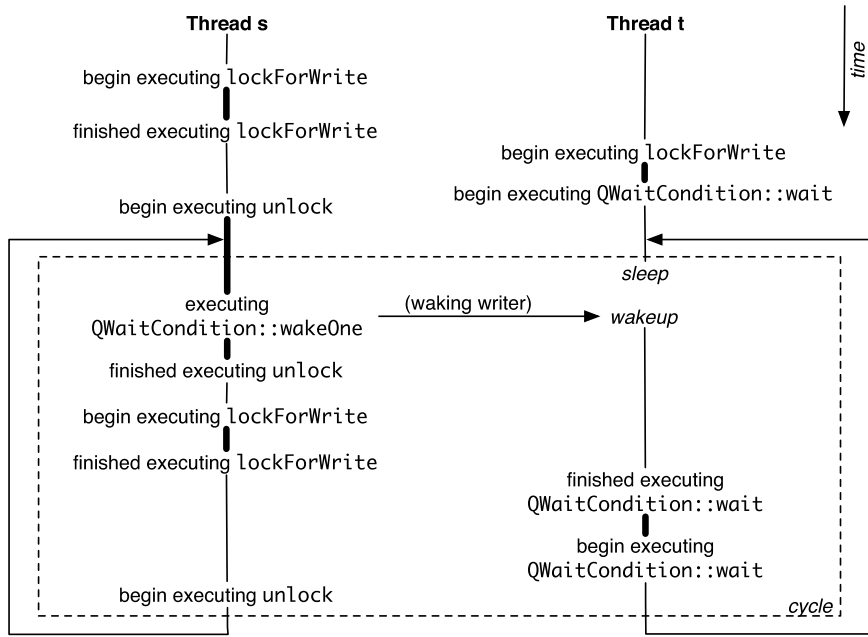


Fig. 9. Graphical representation of the counterexample indicating a starvation problem. The thick black line indicates that the mutex is locked.

4.7. Results

In these experiments we have verified absence of deadlock and starvation and a number of safety properties for a maximum of three threads, and for a maximum of four lock operations. Although the absence of starvation was verified for six lock operations, the safety properties and absence of deadlock were only verified for four lock operations. For these parameters, the experiments runs in about four hours (1:34 for deadlock checking, and 35 min for starvation, and 2:08 for the safety properties), using 127.6 gigabytes of memory. If we increase these values slightly, the execution time worsens drastically and/or the memory usage increases above 128 gigabytes, the memory limit for our machines. So, for a complete correctness result, we have to proceed differently.

5. General reentrant readers–writers model

In this section we will formalise the SPIN model in PVS [24]. We prove that the reentrant algorithm is safe and free from deadlock and writer starvation when we generalise to *any* number of processes. While explaining the formalisation we will briefly introduce PVS.

5.1. Readers–Writers model in PVS

PVS offers an interactive environment for the development and analysis of formal specifications. The system consists of a specification language and a theorem prover. The specification language of PVS is based on classical, typed higher-order logic. It resembles common functional programming languages, such as Haskell, LISP or ML. The choice of PVS as the theorem prover to model the readers–writers locking algorithm is purely based upon the presence of local expertise. The proof can be reconstructed in any reasonably modern theorem prover, for instance Isabelle [23] or Coq [6].

The earlier translation of an Uppaal model of the algorithm to PVS [13] was specific to that particular model. In order to derive the PVS specification from the SPIN model we use a more methodical approach, suitable for other models as well. Furthermore, this methodical approach offers more opportunities for tool support.

There is no implicit notion of state or processes in PVS specifications. So, we construct a state transition system that explicitly keeps track of a system state. This state consists of the global variables of the SPIN model, thread information, and a variable indicating which thread is currently active.

For each thread a program counter and the state of the local variables are also part of the global transition system. Moreover, whether a thread is allowed to be scheduled is kept by means of a `ThreadState`. When it is *Running* the scheduler will allow the thread to progress. However, when it is *Sleeping*, it will not be permitted to run until woken up. A thread can have an *atomic* flag set. This flag tells the scheduler that only this thread can be executed. The *atomic* flag is set whenever the *atomic* primitive is used in SPIN and is reset when the *atomic* block ends. Each critical section in the SPIN model starts with a `QMutex_lock` and ends with a `QMutex_unlock` (e.g. see Fig. 8). These method calls enforce mutual exclusion of

access to all the global variables in the SPIN model. We abstract away from these method calls by setting the `atomic` flag when a thread enters its critical section and resetting the flag once it leaves the critical section. This is semantically the same as using the mutual exclusion mechanism, because threads use only local variables outside of their critical sections.

A thread can transfer its atomic status to another one, say with the `ThreadID` `tid`, by setting the field to `tid`. Only `tid` will be able to be scheduled next.

With `NT` denoting the total number of processes, we get a general representation of threads. What kind of local and global variables are used is left open by means of type parameters. These can be instantiated for each particular SPIN model with a model specific collection of local and global variables.

```
Threads[NT:nat, PC:TYPE, LV:TYPE, GV:TYPE] : THEORY
```

```
BEGIN
```

```
ThreadID : TYPE = below(NT)1
ThreadStateType : TYPE = { Running, Waiting, Terminated }
ThreadState : TYPE = [# state : ThreadStateType
                      , local : LV
                      , PC : PC
                      , atomic : boolean #]2
Threads : TYPE = ARRAY[ ThreadID → ThreadState]3
System : TYPE = [# threads : Threads
                 , currentTID, transfer : ThreadID
                 , global : GV #]
```

```
END Threads
```

The predicate `interleave` simulates parallel execution of threads. A thread is only allowed to switch its context when it is not `atomic` or when the lock is transferred from one thread to another. With `isNull` is tested whether `transfer` contains a valid `ThreadID`. This thread becomes the next current thread. Only `Running` threads are scheduled.

```
interleave(s1,s3:(validState?)) : boolean =
∃ (s2:System) : chain_atomic(s1,s2)
  ∧ IF isNull(s2'transfer)4
    THEN ∃ (tid:ThreadID) : s3 = s2 WITH [ 'currentTID := tid ]5
      ∧ s3'threads(tid)'state = Running
    ELSE s3 = s2 WITH [ 'currentTID := s2'transfer
                      , 'transfer := NT ]
    ENDIF
```

The predicate only holds for a subset of the `System` data type, signified by the `validState?` predicate, further explained in Section 5.3.

Before possibly switching its context, the current thread performs a series of execution steps using the `chain_atomic` relationship. It is assumed that a `next` relation is provided, representing a single step in the execution of a thread. The non-deterministic choice which thread gets to execute is modelled by the existential quantifier that states that any thread can become the next current thread, unless there is an explicit lock transfer.

A single step, as described by the `next` relation, is atomic by definition. A sequence of such steps is executed recursively until the thread has released its atomic flag.

```
chain_atomic(s1:System, s2:System) : RECURSIVE boolean =
  ¬s2'threads(s2'currentTID)'atomic
  ∧ ( next(s1,s2)
    ∨ ∃ (s:System): next(s1,s)
      ∧ s1'currentTID = s'currentTID
      ∧ s1'threads(s1'currentTID)'atomic
      ∧ s1'threads(s1'currentTID)'atomic = s'threads(s'currentTID)'atomic
      ∧ chain_atomic (s,s2)
  )
MEASURE s1 BY state_order
```

This recursive relationship terminates because there are no cycles in the progression of states a thread can transfer to with its atomic flag set.

The SPIN model also makes use of synchronisation primitives in order to put threads to sleep and wake them up using the `QWaitCondition` class. The PVS specification used here is semantically slightly different from the one used in QT. This

¹ Denotes the set of natural numbers between 0 and `NT`, exclusive of `NT`.

² Recordtypes in PVS are surrounded by `[#` and `#]`.

³ Arrays in PVS are denoted as functions.

⁴ The `'` operator denotes record selection.

⁵ The `'` operator can also be used on default values, in this case `s2`.

model not only wakes up a process, but also passes the lock on with the `transfer` field to one of the woken threads to avoid writer starvation, mentioned as a solution to the starvation problem at the end of Section 4.6. Note that this is only possible if a thread immediately leaves its critical section after synchronisation. The model is based on a FIFO queue that holds all processes, such that they will be woken in the order that they have been put to sleep.

```
QWaitCondition : TYPE = list[ThreadID]
NEQWaitCondition : TYPE = {wc:QWaitCondition | length(wc) > 0 }6

wait(s:System, q:QWaitCondition) : [System, QWaitCondition] =
  (s WITH [ 'threads(s'currentTID)'state := Waiting
            , 'threads(s'currentTID)'atomic := false ]
    , append(q, cons(s'currentTID, null)))

wakeOne(s:System, q:NEQWaitCondition) : [System, QWaitCondition] =
  (s WITH [ 'threads(car(q))'state := Running
            , 'threads(s'currentTID)'atomic := false
            , 'threads(car(q))'atomic := true
            , 'transfer := car(q)], cdr(q))

wakeAll(s:System, q:NEQWaitCondition) : [System, QWaitCondition] =
  LET newthreads = λ (p:ThreadID) : s'threads(p)
    WITH [ state := IF member(p,q)
          THEN Running
          ELSE s'threads(p)'state
        ENDIF ] IN
  (s WITH [ 'threads := newthreads
            , 'threads(s'currentTID)'atomic := false
            , 'threads(car(q))'atomic := true
            , 'transfer := car(q)], null)
```

The type `QWaitCondition` is a list that holds the `ThreadID`s of all threads that are put to sleep. The `wait` function takes a wait queue and changes the state of the current thread to `Waiting` and releases the `atomic` flag.

The `wakeOne` and `wakeAll` functions are used to wake up one waiting writer and all waiting readers respectively. Their states are set to `Running` so they can be scheduled and the lock is transferred to the process that is first in the queue.

5.2. Translation from spin to PVS

After having defined all the components, the total state of the model is defined by all the local and global variables. These are exactly the same as in the original SPIN model as defined in Fig. 8. The `ProgramCounterStates` refer to the locations of the program counter as the SPIN model executes. For instance, the start of the outer do loop in the `user()` function defined in Fig. 6 contributes `user05` to `ProgramCounterStates`.

`ProgramCounterStates` instantiates `PC` in the theory `Threads` and similarly, both `LocalVariables` and `GlobalVariables` instantiate `LV` and `GV` respectively.

```
ProgramCounterStates : TYPE = { lockForRead17, ... , user05 }7
LocalVariables : TYPE = [# readNest, writeNest, maxLocks : nat #]
QReadWriteLock : TYPE =
  [# readerWait, writerWait : QWaitCondition
   , count : [ThreadID → nat]
   , currentWriter : ThreadID
   , threadCount, waitingReaders, waitingWriters : nat #]
GlobalVariables : TYPE =
  [# readers, writers : nat, rlock : QReadWriteLock #]
```

The relation `next(s1,s2 : System) : boolean` specifies the global state transitions.

The body of this function is derived directly from the SPIN model using the following method.

- At each position where there can be a context switch in the SPIN model, there is a location added to the program counter type.
- Non-deterministic choices are modelled as disjunctions in the transition relation. There is one disjunct for each non-deterministic choice.

⁶ The `NEQWaitCondition` type prevents the use of wake functions on empty queues.

⁷ Most locations are omitted for brevity.

- Control structures like `do` are translated by setting the program counter to the appropriate location. Location labels are derived from the function names, appended with the line numbers in the SPIN source code. In principle, numbers only would have sufficed, but for readability reasons the function name was added.
- Function calls are done by setting the program counter to the location of the function. Since no function is called from more than one location, using a return address or even using a stack for more than one return address has been omitted.
- Assignments are translated to modifications of the local or global variables in the state.

The auxiliary variables `readNest`, `writeNest` and `MAXLOCKS` restrict the SPIN model to a maximum number of nested reads and writes. They also prevent unwanted sequences of lock/unlock operations, e.g. when a write lock request occurs after a read lock has already been obtained. This `user()` function from Fig. 6 is directly coded in the state transition model, where each label corresponds to the position of the program counter in the original.

```
next(s1:System, s2:System) : boolean =

[ .. removed some code for brevity .. ]

CASES s1'threads(currentTID)'PC OF
user05:
    s2 = s1 WITH [ 'threads(currentTID)'local'maxLocks := MAXLOCKS8
                  , 'threads(currentTID)'PC := user07 ],
user07:
    ( IF maxLocks > 09
      THEN s2 = s1 WITH [ 'threads(currentTID)'local'maxLocks := maxLocks-1
                        , 'threads(currentTID)'PC := user10]
      ELSE FALSE ENDIF
    ∨ IF writeNest + readNest > 0
      THEN s2 = s1 WITH [ 'threads(currentTID)'PC := unlock67]
      ELSE FALSE ENDIF
    ∨ IF writeNest + readNest = 0
      THEN s2 = s1 WITH [ 'threads(currentTID)'PC := user05 ]
      ELSE FALSE ENDIF ),
user10:
    ( IF readNest = 0
      THEN s2 = s1 WITH [ 'threads(currentTID)'PC := lockForWrite42]
      ELSE FALSE ENDIF
    ∨ s2 = s1 WITH [ 'threads(currentTID)'PC := lockForRead17] ),

[ .. transition relation continues with cases for lockForRead, etc. .. ]
```

As an example we provide the transition model derived from the SPIN code in Fig. 8 for the `lockForRead` function by using the rules specified earlier.

```
CASES s1'threads(currentTID)'PC OF

[ .. removed some code for brevity .. ]

lockForRead17:
    s2 = s1 WITH [ 'threads(currentTID)'atomic := true %QMutexLock
                  , 'threads(currentTID)'PC := lockForRead18 ],
lockForRead18:
    IF count(currentTID) = 010
    THEN s2 = s1 WITH [ 'threads(currentTID)'PC := lockForRead20 ]
    ELSE s2 = s1 WITH [ 'threads(currentTID)'PC := lockForRead35 ]
    ENDIF,
lockForRead20:
    IF currentWriter ≠ NT ∨ waitingWriters > 0
    THEN LET s = s1 WITH [ 'global'rwlock'waitingReaders
                        := waitingReaders + 1
                        , 'threads(currentTID)'PC := lockForRead26 ]
    IN LET (s_upd,q_upd) = wait(s, readerWait)
```

⁸ In some places `s'variableName` or `s'global'rwlock'variableName` is abbreviated to `variableName`.

⁹ Could be replaced with `maxLocks > 0 ∧ s2 = s2 ..`, but the if construction is maintained to show correspondence with the promela model.

¹⁰ All references to global variables should be read as being prefixed by `s1'global'rwlock`. I.E. `count(currentTID)` abbreviates `s1'global'rwlock'count(currentTID)`.

```

      IN s2 = s_upd WITH [ 'global'rwlock'readerWait := q_upd ]
    ELSE %-(s1'global'currentWriter ≠ NT) ∨ s1'global'waitingWriters > 0)
      s2 = s1 WITH [ 'threads(currentTID)'PC := lockForRead31 ]
    ENDIF,
lockForRead26:
  s2 = s1 WITH [ 'global'rwlock'waitingReaders := waitingReaders - 1
                , 'threads(currentTID)'PC := lockForRead20 ],
lockForRead31:
  s2 = s1 WITH [ 'global'rwlock'threadCount := threadCount + 1
                , 'threads(currentTID)'PC := lockForRead35 ],
lockForRead35:
  S2 = s1 WITH [ 'global'rwlock'count(currentTID)
                := count(currentTID) + 1
                , 'threads(currentTID)'atomic := false
                , 'threads(currentTID)'PC := incReadNest01 ],

```

After obtaining a read lock, the variable `readNest` has to be increased, corresponding with the code that updates the model variables in the original SPIN model.

The transition model starts out with all threads in a `Running` state and with the local variables at their initial values. Also, the global variables are all initialized and all queues are empty.

```

starting? : PRED[ThreadState] = { (t:ThreadState) | t.state = Running ∧
¬t'atomic ∧ t'PC = user05 ∧ t'local'readNest = 0 ∧ t'local'writeNest = 0 }

```

```

startingState(s1:System) : bool = threadCount = 0
∧ currentWriter = NT ∧ waitingReaders = 0 ∧ waitingWriters = 0
∧ readerWait = null ∧ writerWait = null ∧ readers = 0 ∧ writers = 0
∧ ∀ (tid:ThreadID) :
  ( starting?(s1'threads(tid)) ∧ s1'global'rwlock'count(tid) = 0 )

```

5.3. System invariants

In a system state, not every combination of variables will be reached during normal execution of the program. A certain amount of redundancy is present in the set of variables in the model. For instance, the number of writers waiting can be deduced both from the `waitingWriters` variable as well as the length of the wait queue. Also, variables are maintained that keep track of the total amount of processes that occupy the critical section and of the number of processes that are waiting for a lock. We express the integrity of the values of those variables by using a `validState?` predicate. This is an invariant on the global state of all the processes and essential in proving that the algorithm is deadlock free. We want to express in this invariant that the global state is sane and safe at the time a context switch can take place. Sanity is defined as:

- The value of the `waitingReaders` should be equal to the total number of processes with a status of `Waiting` and that are a member of the `readerWait` queue. Counting the members of the wait queue is done by the recursive `waitingReaders` function.

```

waitReadInv(s:System) : bool =
  s'global'rwlock'waitingReaders = waitingReaders(s)

```

- The value of the `waitingWriters` should be equal to the total number of processes with a status of `Waiting` and that are a member of the `writerWait` queue. The `waitingWriters` function counts the waiters in the queue.

```

waitWriteInv(s:System) : bool =
  s'global'rwlock'waitingWriters = waitingWriters(s)

```

- The value of the `threadCount` variable should be equal to the number of processes with a lock count of 1 or higher and at the same time this equals the total number of `readers` and `writers`. Again, recursively defined in the `count` function.

```

countInv(s:System) : bool =
  s'global'rwlock'threadCount = count(s'threads)

```

Besides the redundant variables having sane values, we also prove that the invariant implies that a waiting process does not have a lock, indicated by having a count of zero, stored in the `count` array for that particular process. If it has obtained a lock, it must necessarily be `Running`.

```

statusInv(s:System) : bool = ∀(tid:ThreadID): LET thr = s'threads(tid) IN
  thr.state = Waiting ⇒ s'global'rwlock'count(thr) = 0
∧ s'global'rwlock'count(thr) > 0 ⇒ thr.state = Running

```

Part of the invariants defined in Section 5.3 are defined as `safetyInv` and proven as well.


```
safetyInv(s:System) : bool =
  (readers = 0 ∨ writers = 0) ∧ writers ≤ 1
```

Furthermore, if a process has obtained a write lock, then only that process can occupy the critical section:

```
writeLockedByInv(s:System) : bool = currentWriter ≠ NT ⇒ threadCount = 1 ∧
  count(currentWriter) > 0 ∧
  ∀(tid:ThreadID): tid ≠ currentWriter ⇒ count(tid) = 0))
```

The combination of all these invariants makes up a valid state.

```
validState?(s:System) : bool = countInv(s) ∧ waitWriteInv(s) ∧
  statusInv(s) ∧ writeLockedByInv(s) ∧ safetyInv(s) ∧ waitReadInv(s)
```

The definition of `interleave` generates a type correctness condition that will guarantee that if we are in a valid state, as defined by the `validState?` predicate, we will transition with an interleaving to another state that is still valid. We also show that the starting state is a valid state. The proof of this correctness condition is a straightforward, albeit large, case distinction with the help of some auxiliary lemmas.

5.4. Freedom from deadlocks and livelocks

The theorem-prover PVS does not have an innate notion of deadlock. If, however, we consider the state transition model as a directed graph, in which the edges are determined by the `interleave` function, deadlock can be determined by identifying states in the state transition graph having no outgoing edges. This interpretation of deadlock, however, can be too limited. If, for example, there is a situation where a process alters one of the state variables in a non-terminating loop, a deadlock will not be detected, because each state has an outgoing edge. There still can be *livelock*; transitions are possible, but there will be no progress. To prove there can be no livelock, we define a well-founded ordering on the all valid system states and show that for each state reachable from the starting state (except for the starting state itself), there exists a transition to a smaller state according to that ordering. The smallest element within the order is the starting state. This means that for each reachable state there exists a path back to the starting state and consequently it is impossible for any process to get stuck in a such a loop indefinitely. Moreover, this also covers the situation in which we would have a *local deadlock* (i.e. several but not all processes are waiting for each other).

We create a well-founded ordering by defining a state to become smaller if the number of waiting processes decreases or alternatively, if the number of waiting processes remains the same and the total count of the number of processes that have obtained a lock is decreasing. Well foundedness follows directly from the well foundedness of the lexicographical ordering on pairs of natural numbers.

```
smallerState(s2, s1 : (validState?)) : bool =
  numberWaiting(s2) < numberWaiting(s1) ∨
  numberWaiting(s2) = numberWaiting(s1) ∧ totalCount(s2) < totalCount(s1)
```

The `numberWaiting` function is a function on the array of thread-states that yields the number of processes that have a `Waiting` status. The `totalCount` function computes the sum of all the elements of the `count` array.

Once we have established that each state transition maintains the `validState?` invariant, all we have to prove is that each transition has outgoing states and that all of these states (except for the starting state) will possibly result in a state that is smaller. This is the `noDeadlock` theorem.

```
noDeadlock: THEOREM
  ∀(s1: (validState?)) : ∃(s2: (validState?)) : interleave(s1, s2)
  ∧ (¬startingState(s1) ⇒ smallerState(s2, s1))
```

All that is needed to prove this theorem is a case distinction and inductive proofs of auxiliary lemmas that state that the recursively defined counting functions used in the invariant definitions are only decreased and increased if certain preconditions are met.

The proofs of the absence of deadlock proceeds analogously to the proof that was done for in earlier more abstract version of this model by the same authors [13].

5.5. Freedom from starvation

There is no built-in notion of starvation in PVS either. We define the absence of starvation as a theorem stating that if a thread intends to acquire a lock, it will eventually obtain it. The intention is identified by the thread entering the `lockForWrite` part of the code.

```
noWriterStarvation: THEOREM
  ∀ (s1:(validState?)) : s1'threads(s1'currentTID)'PC = lockForWrite42
  ⇒ lock_on_trace(s1, s1'currentTID)
```

Eventually obtaining the lock is defined using the observation that for all traces of possible interleaves, the thread that signalled the intention to acquire a lock will become the current writer.

```
lock_on_trace(s1:System, lockTID:ThreadID) : RECURSIVE boolean =
  ∀ (s2:(ValidState?)) : interleave(s1,s2)
  ∧ ( s2'global'rwlock'currentWriter = lockTID ∨ lock_on_trace(s2, lockTID))
MEASURE s1 BY lock_on_trace_measure(lockTID)
```

This recursive relationship is well founded, since the measure defined in this function guarantees termination. Proving that for each interleaving the measure decreases, again, is done by a massive case distinction. The complete proof, including the proof of the absence of writer starvation is available at <http://www.cs.ru.nl/~sjakie/papers/readerswriters/> where also the original code and the SPIN models can be found.

All together, the derivation of the PVS model, the determination of the invariants as well as proving the theorems and auxiliary lemmas took one of the authors about a month and a half.

6. Related and future work

Several studies investigated *either* the conversion of code to state transition models, as is done e.g. in [11] with mcrl2 or the transformation of a state transition model specified in a model checker to a state transition model specified in a theorem prover, as is done e.g. in [20] using VeriTech. With the tool TAME one can specify a time automaton directly in the theorem prover PVS [3]. For the purpose of developing consistent requirement specifications, the transformation of specifications in a model checker (Uppaal [21]) to specifications in PVS has been studied in [10].

In [25] model checking and theorem proving are combined to analyze the classic non-reentrant (in contrast to the reentrant version studied in our paper) readers–writers problem. The authors do not start with actual industrial source code but they start from a tabular specification that can be translated straightforwardly into SPIN and PVS. Safety and clean completion properties are derived semi-automatically.

[17] reports on experiments in combining theorem proving with model checking for verifying transition systems. The complexity of systems is reduced abstracting out sources for unboundedness using theorem proving, resulting in a bounded system suited for being model checked.

The verification framework SAL [28] combines different analysis tools and techniques for analysing transition systems. Besides model checking and theorem proving it provides program slicing, abstraction and invariant generation.

In [15] part of an aircraft control system is analyzed, using a theorem prover. On a single configuration this was previously studied with a model checker. A technique called *feature-based decomposition* is proposed to determine inductive invariants. It appears that this approach admits incremental extension of an initially simple base model making it better scalable than traditional techniques.

Java Pathfinder (JPF) [32] operates directly on Java making a transformation of source code superfluous. If the code studied would have been written in Java, JPF would have been the foremost candidate tool for this case study. This can be done directly within JPF or, if that is desirable, even by generating PROMELA code as was done originally in [16]. It would be interesting to compare the effort, ease of modelling and ease/performance of model checking of tools for different languages by taking the case study of this paper and performing it also for the same algorithm written in Java using e.g. the extension of JPF with symbolic execution [1]. Alternatively, Bandera [8] could be used for such a comparative case study. Bandera includes support for abstractions which may be very useful in such a case study. It translates Java programs to the input languages of SMV and SPIN. There is an interesting connection between Bandera and PVS. To express that properties do not depend on specific values, Bandera provides a dedicated language for specifying abstractions, i.e. concrete values are automatically replaced by abstract values, thus reducing the state space. The introduction of these abstract values may lead to prove obligations which can be expressed and proven in PVS.

In [27] a model checking method is given which uses an extension of JML [22] to check properties of multi-threaded Java programs.

With Zing [2] on the one hand models can be created from source code and on the other hand executable versions of the transition relation of a model can be generated from the model. This has been used successfully by Microsoft to model check parts of their concurrency libraries.

Future work

The methodology used (creating in a structured way a model close to the code, model checking it first and proving it afterwards [29]) proved to be very valuable. We found a bug, improved the code, extended the capabilities of the code and proved it correct. One can say that the model checker was used to develop the formal model which was proven with the theorem prover. This decreased significantly the time investment of the use of a theorem prover to enhance reliability. However, every model was created manually. We identified several opportunities for tool support and further research.

Bounded model related to source code. Tool support could be helpful here: not only to ‘translate’ the code from the source language to the model checker’s language. It could also be used to record the abstractions that are made. In this case

that were: basic locks → lock process model, hash tables → arrays, threads → processes and some name changes. A tool that recorded these abstractions, could assist in creating trusted source code from the model checked model.

Deep versus shallow embedding. A complete specification of the semantics and syntax of PROMELA in PVS was avoided in our construction of the PVS model. We focused on methodically translating between the two models. Greater confidence of the translation may be achieved by using a translation that preserves the structure of the original PROMELA code instead.

Relation of finite to unbounded model. It would be interesting to prove that the model in the theorem prover and the model checked are properly related, e.g. by establishing a refinement relation [4] between them. Interesting methods to do this would be using a semantic compiler, as was done in the European Robin project [31], or employing a specially designed formal library for models created with a model checker, e.g. TAME [3].

Relation of unbounded model to source code. Another interesting future research option is to investigate generating code from a fully proven PVS model. This could be code generated from code-carrying theories [19] or it could be proof-carrying code through the use of refinement techniques [4].

7. Concluding remarks

We have investigated Trolltech's widely used industrial implementation of the reentrant readers–writers problem. Model checking revealed an error in the implementation (version 4.3). Trolltech was informed about the bug. Recently, Trolltech released a new version of the thread library (version 4.4) in which the error was repaired. However, the new version of the Qt library is still only weakly reentrant, not admitting threads that have write access to do a read lock. This limitation unnecessarily hampers modular programming.

The improved readers–writers model described in this paper is *deadlock free* and *strongly reentrant*. The model was first developed and checked for a limited number of processes using a model checker. Then, the properties were proven for any number of processes using a theorem prover. We also studied the *absence of starvation*. With model checking a starvation problem was revealed. We created a starvation free implementation and checked it with model checking. We have sketched the outline of a proof for that implementation.

Acknowledgements

We want to thank the reviewers of this paper for their helpful advice and their constructive comments and corrections. They helped us to considerably improve our paper.

References

- [1] S. Anand, C.S. Pasareanu, W. Visser, JPF-SE: a symbolic execution extension to Java PathFinder, in: O. Grumberg, M. Huth (Eds.), TACAS, in: Lecture Notes in Computer Science, vol. 4424, Springer, 2007, pp. 134–138.
- [2] T. Andrews, S. Qadeer, S.K. Rajamani, J. Rehof, Y. Xie, Zing: a model checker for concurrent software, in: R. Alur, D. Peled (Eds.), CAV, in: Lecture Notes in Computer Science, vol. 3114, Springer, 2004, pp. 484–487.
- [3] M. Archer, C. Heitmeyer, S. Sims, TAME: a PVS interface to simplify proofs for automata models, in: User Interfaces for Theorem Provers, Eindhoven, The Netherlands, 1998.
- [4] M.A. Barbosa, A refinement calculus for software components and architectures, SIGSOFT Softw. Eng. Notes 30 (5) (2005) 377–380.
- [5] M. Ben-Ari, Principles of the Spin Model Checker, Springer, 2008.
- [6] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, in: Texts in Theoretical Computer Science, Springer Verlag, 2004.
- [7] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite state concurrent systems using temporal logic specifications: a practical approach, in: POPL, 1983, pp. 117–126.
- [8] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby H. Zheng, Bandera: extracting finite-state models from java source code, in: Software Engineering, 2000. Proceedings of the 2000 International Conference on, 2000, pp. 439–448.
- [9] P.J. Courtois, F. Heymans, D.L. Parnas, Concurrent control with “readers” and “writers”, Commun. ACM 14 (10) (1971) 667–668.
- [10] A. de Groot, Practical automaton proofs in PVS, Ph.D. thesis, Radboud University Nijmegen, 2008.
- [11] M. van Eekelen, S. ten Hoedt, R. Schreurs, Y.S. Usenko, Analysis of a session-layer protocol in mCRL2. Verification of a real-life industrial implementation, in: P. Merino, S. Leue (Eds.), Proc. 12th Int'l Workshop on Formal Methods for Industrial Critical Systems, FMICS 2007, in: Lecture Notes Computer Science, vol. 4916, Springer, 2008, pp. 182–199.
- [12] B. van Gastel, Verifying reentrant readers–writers. Master's thesis, Radboud Universiteit, Nijmegen, Netherlands, March 2010.
- [13] B. van Gastel, L. Lensink, S. Smetsers, M. van Eekelen, Reentrant readers–writers: a case study combining model checking with theorem proving, in: D. Cofer, A. Fantechi (Eds.), Formal Methods for Industrial Critical Systems: 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15–16, 2008, Revised Selected Papers, in: Lecture Notes Computer Science, vol. 5596, Springer-Verlag, 2009, pp. 85–102. Received the EASST FMICS2008 Best Software Science Paper Award.
- [14] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, D. Lea, Java Concurrency in Practice, Addison Wesley Professional, 2006.
- [15] V. Ha, M. Rangarajan, D. Cofer, H. Rues, B. Dutertre, Feature-based decomposition of inductive proofs applied to real-time avionics software: an experience report, in: ICSE'04: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2004, pp. 304–313.
- [16] K. Havelund, T. Pressburger, Model checking Java programs using Java PathFinder, STTT 2 (4) (2000) 366–381.
- [17] K. Havelund, N. Shankar, Experiments in theorem proving and model checking for protocol verification, in: M.-C. Gaudel, J. Woodcock (Eds.), FME'96: Industrial Benefit and Advances in Formal Methods, Springer-Verlag, 1996, pp. 662–681.
- [18] G. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, Boston, USA, 2004.
- [19] B. Jacobs, S. Smetsers, R. Wichers Schreur, Code-carrying theories, Form. Asp. Comput. 19 (2) (2007) 191–203.

- [20] S. Katz, Faithful translations among models and specifications, in: FME'01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity, Springer-Verlag, London, UK, 2001, pp. 419–434.
- [21] K.G. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, *Int. J. Softw. Tools Technol. Transfer* 1 (1–2) (1997) 134–152.
- [22] G.T. Leavens, J.R. Kiniry, E. Poll, A JML tutorial: modular specification and verification of functional behavior for Java, in: W. Damm, H. Hermanns (Eds.), CAV, in: *Lecture Notes in Computer Science*, vol. 4590, Springer, 2007, p. 37.
- [23] T. Nipkow, L.C. Paulson, M. Wenzel, Isabelle/HOL—A Proof Assistant for Higher-Order Logic, in: LNCS, vol. 2283, Springer, 2002.
- [24] S. Owre, J.M. Rushby, N. Shankar, PVS: a prototype verification system, in: D. Kapur (Ed.), 11th International Conference on Automated Deduction, CADE, in: *Lecture Notes in Artificial Intelligence*, vol. 607, Springer-Verlag, Saratoga, NY, 1992, pp. 748–752.
- [25] V. Pantelic, X.-H. Jin, M. Lawford, D.L. Parnas, Inspection of concurrent systems: combining tables, theorem proving and model checking, in: H.R. Arabnia, H. Reza (Eds.), *Software Engineering Research and Practice*, CSREA Press, 2006, pp. 629–635.
- [26] J.-P. Queille, J. Sifakis, Specification and verification of concurrent systems in cesar, in: M. Dezani-Ciancaglini, U. Montanari (Eds.), *Symposium on Programming*, in: *Lecture Notes in Computer Science*, vol. 137, Springer, 1982, pp. 337–351.
- [27] Robby E. Rodríguez, M.B. Dwyer, J. Hatcliff, Checking JML specifications using an extensible software model checking framework, *STTT* 8 (3) (2006) 280–299.
- [28] N. Shankar, Combining theorem proving and model checking through symbolic analysis, in: C. Palamidessi (Ed.), CONCUR, in: *Lecture Notes in Computer Science*, vol. 1877, Springer, 2000, pp. 1–16.
- [29] S. Smetsers, M. van Eekelen, LaQuSo: using formal methods for analysis, verification and improvement of safety critical software, in: Pedro Merino and Erwin Schoitsch (Eds.) *ERCIM News, Special Theme Safety-Critical Software*, vol. 75, October 2008, pp. 38–39.
- [30] H. Sutter, The free lunch is over: a fundamental turn toward concurrency in software, *Dr. Dobbs's J.* 30 (3) (2005).
- [31] H. Tews, T. Weber, M. Völz, E. Poll, M. van Eekelen, P. van Rossum, Nova micro-hypervisor verification. Technical Report ICIS-R08012, Radboud University Nijmegen, May 2008. Robin deliverable D13.
- [32] W. Visser, K. Havelund, G.P. Brat, S. Park, F. Lerda, Model checking programs, *Autom. Softw. Eng.* 10 (2) (2003) 203–232.